Implementação eficiente do algoritmo Keyak para ARMv8

André C. de Moraes[1], **Diego F. Aranha**[1]

[1]Instituto de Computação - Universidade Estadual de Campinas (UNICAMP)

andre.moraes@students.ic.unicamp.br, dfaranha@ic.unicamp.br

Abstract. *Keyak is an authenticated encryption scheme based on sponge construction. Cryptographic sponges provide a good amount of security at a high computational cost, which hinders its performance, specially in resource-constrained devices. The main contributions of this work aimed to improve the performance of the Keyakv2 in platforms using the ARM architecture, by implementing the algotrithm's core functions in assembly language with the NEON vector instruction set.*

Resumo. *Keyak é um algoritmo de cifração autenticada baseada na construção esponja. Esponjas criptográficas fornecem um bom nível de segurança por um custo computacional elevado, o que prejudica seu desempenho, especialmente em dispositivos com recursos limitados. As contribuições deste trabalho objetivaram aprimorar o desempenho do Keyakv2 nas plataformas baseadas na arquitetura ARM através da implementação em linguagem de montagem das principais rotinas do algoritmo utilizando com o conjunto de instruções vetoriais NEON.*

1. Introdução

Cifração autenticada é um algoritmo que providencia confidencialidade e autenticidade de dados, onde a decifração é feita de maneira conjunta com a verificação de integridade [K. Paterson 2013]. Tais propriedades protegem o algoritmo de ataques sob o cenário de *Man-in-the-Middle* [F. Callegati et al 2009], como *bit-flipping* [K. Graves 2010].

Um padrão muito utilizado para esse tipo de algoritmo é conhecido como **construção esponja**, que possui propriedades relevantes para a cifração autenticada como tamanho de entrada e saída arbitrários [G. Bertoni et al. 2011a]. **Keccak** é uma família de algoritmos de construções esponja, utilizada em diversas aplicações criptográficas como cifração autenticada, função *hash* e gerador de números aleatórios [G. Bertoni et al. 2013a]. Como destaque, o algoritmo **SHA-3**, homologado pelo *NIST*, faz parte da família *Keccak* [NIST 2014].

Keyak [G. Bertoni et al. 2015a] é um algoritmo baseado na construção esponja conhecida como **Keccak-p** [G. Bertoni et al. 2013a], proposta pelos mesmos autores do *Keccak*, e é uma das submissões para a competição **CAESAR**. A competição busca promover um novo algoritmo padrão na área de cifração autenticada. O **Keyakv2**, segunda versão da primitiva *Keyak*, cumpre todos os requisitos estabelecidos na terceira e mais recente especificação da competição [G. Bertoni et al. 2015b]. Qualquer menção no texto ao algoritmo *Keyak* a partir de então refere-se ao *Keyakv2*.

Neste trabalho, buscamos desenvolver uma implementação de alto desempenho do *Keyak* para dispositivos com **ARMv8**, a ser explicada em detalhes na **Seção 2**. O

código desenvolvido [A. Moraes 2016] foi aceito pelos autores do *Keyak* e pode ser visto no repositório oficial [G. Bertoni et al. 2015d]. Durante os testes, foi possível notar um erro na implementação em *assembly* do *ARMv7*, e notificar os autores do *Keyak* com a correção necessária [A. Moraes 2016].

2. Proposta

O algoritmo *Keyak* fornece um bom nível de segurança na troca de dados, sendo uma solução válida para desenvolvedores de *software* que busquem prover segurança para suas aplicações. Entretanto, sua baixa eficiência dificulta o seu uso em dispositivos com recursos limitados, como telefones celulares e *tablets*.

Uma maneira de contornar essa restrição é implementar as rotinas mais custosas do algoritmo em línguagem de máquina otimizada para as principais arquiteturas do mercado. No repositório oficial [G. Bertoni et al. 2015d] dos autores do *Keyak*, constam implementações para diversas arquiteturas, como por exemplo *ARMv7*.

A nova geração de processadores *ARMv8* introduz várias novidades em relação à geração antiga *ARMv7*, como o processador operar em 64 *bits* e não mais 32 *bits*, o pacote instruções vetoriais *NEON* tornando-se padrão e não mais opcional, uma quantidade maior de registradores vetoriais e uma nova síntaxe de linguagem de máquina. Entretanto tais mudanças acabam deixando a arquitetura não compatível com código legado de gerações anteriores.

Em vista desses fatos, é necessário escrever uma nova implementação para a arquitetura *ARMv8*, utilizando suas novas capacidades. Neste trabalho, procurou-se utilizar de instruções vetoriais sempre que possível, em oposição à implementação existente do *ARMv7* que as utiliza apenas em alguns casos.

3. Fundamentação Teórica

3.1 Construção Esponja

A **construção esponja** [G. Bertoni et al. 2011a] é definida como uma máquina de estados finita que consome dados de entrada de tamanho arbitrário, resultando em uma saída de um outro tamanho arbitrário. A esponja possui um **estado interno** de **largura** b *bits*, composta de duas partes, o ***bitrate*** de tamanho r a **capacidade** de tamanho c, sendo $b = c + r$. A esponja utiliza de uma **função de permutação** f que é aplicada em todos os b *bits* do seu estado interno.

Durante a fase de absorção, a esponja insere *bits* da entrada através de operações de ***XOR*** com os dados já existentes no estado, aplicando a função f a cada r *bits* inseridos. Após todos os dados da entrada serem inseridos no estado, a esponja entra na fase de compressão, e *bits* são extraídos do estado, aplicando f a cada r *bits* extraídos. A quantidade c de *bits* correspondente à capacidade do estado não é relacionada diretamente com os dados de entrada ou saída, sendo apenas um resultado de aplicações de f sobre o estado interno.

A única restrição sobre os dados de entrada e saída é que eles devem estar particionados em **blocos** $\sigma_0, \sigma_1, ..., \sigma_n$ de tamanho W *bits* cada (chamado de unidade de alinhamento), sendo r um múltiplo de W, o que é facilmente obtido utilizando alguma regra de preenchimento (*padding*).

3.2 Construção Duplex

A **construção duplex** [G. Bertoni et al. 2011b] é um caso particular de uma construção esponja em que não existe diferença entre a fase de absorção e a fase de compressão, uma vez que dados são lidos da entrada e produzidos na saída junto com cada aplicação da função de permutação f sobre o estado, em uma única fase de operação denominada ***duplexing***. O nível de segurança continua equivalente, porém há um aumento considerável do desempenho uma vez que o número de execuções da função de permutação f cai consideravelmente.

Uma **construção duplex de estado total** (*FDS*) é um caso particular de construção *duplex*, onde o *bitrate* de absorção passa a ser o tamanho total do estado, ou seja $b = r$, aumentando o número de *bits* inseridos e consequentemente o desempenho, sem custos significativos para a segurança [B. Mennink et al 2015], uma vez que a saída não contém *bits* da capacidade.

A **construção duplex de estado total chaveada** (*FSKD*) por sua vez é obtida quando se inicializa uma **FDS** com uma chave K de tamanho k e então se executa a função f antes de inserir qualquer dado da entrada. É possível inserir apenas a chave ou inserir a chave e inicializar os $(b - k)$ *bits* restantes com os primeiros blocos σ_i da entrada. Em ambos os casos, é necessário garantir que a chave também possa ser particionada em blocos de mesmo tamanho W que a entrada, através do uso de alguma regra de preenchimento.

A **Figura 1** representa uma construção duplex de estado total chaveada, onde a chave inserida é denominada K, a entrada é inserida em blocos σ_i, e a saída é obtida em blocos Z_i.

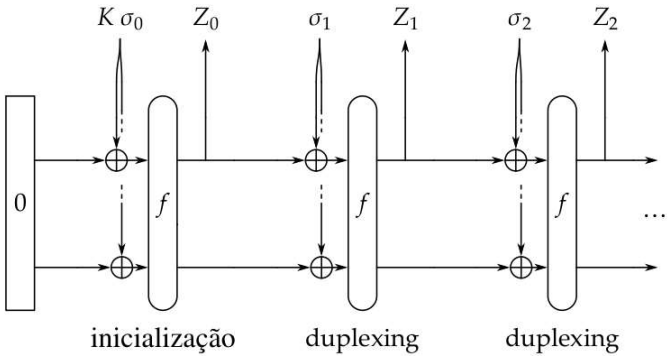

Figura 1. Exemplo de um *FKSD* - [G. Bertoni et al. 2011b]

3.3 Keccak

A família de esponjas **Keccak** utilizam-se de uma mesma função de permutação f conhecida como **Keccak-f**. Existem diversas variações da **Keccak-$f[b]$** quanto ao tamanho b *bits* aceito, sendo necessário que $b = 25 * 2^l$ com $0 \leq l \leq 6$, ou seja, $b \in (25, 50, 100, 200, 400, 800, 1600)$ *bits*. Esse requisito é uma consequência do algoritmo, que requer mapear o estado recebido para um vetor de permutação de **25 posições**, cada qual de tamanho 2^l *bits*, para então realizar operações neste vetor e finalmente retornar o vetor de permutação para o estado resultante.

As operações se resumem a cinco transformações internas [J.P. Aumasson et al 2009], θ, ρ, π, χ e ι, que são executadas um número N_r de vezes, que depende do tamanho b através da relação $N_r = 12 + 2l$. As transformações envolvem operações lógicas, rotações e permutações entre diversos elementos do vetor de permutação.

Keccak-p é uma função de permutação [G. Bertoni et al. 2013a] que consiste na execução **das últimas n_r rodadas** da função de permutação *Keccak-f* associada, portanto $n_r \leq N_r$. No caso particular em que $n_r = N_r$, temos que *Keccak-$p[b, n_r]$* = *Keccak-$f[b]$*. A razão de se executar apenas algumas rodadas se dá pelo fato de que não é necessário realizar todas as rodadas possíveis para se obter um bom nível de segurança e devido ao custo elevado de cada rodada adicional.

4. Keyak

Keyak é um algoritmo composto de esponjas internas, onde a função de permutação é necessariamente alguma variação da *Keccak-$p[b, n_r]$*, de estado b *bits* e n_r rodadas. Em particular, os autores do *Keyak* recomendam cinco instâncias nomeadas: ***River Keyak***, ***Lake Keyak***, ***Sea Keyak***, ***Ocean Keyak*** e ***Lunar Keyak***. Todas possuem mesma capacidade $c = 256$, tamanho de rótulo $\tau = 128$ e número de rodadas $n_r = 12$. As instâncias se diferem no tamanho da largura b, o qual é 800 para *River* e 1600 para as demais, e ao número de esponjas com *River* e *Lake* possuindo apenas uma única instância, e *Sea*, *Ocean* e *Lunar* possuindo duas, quatro e oito instâncias respectivamente.

O *Keyak* opera em cima do chamado modo ***Motorist***, que é uma construção [G. Bertoni et al. 2015a] baseada em *FSKD* para cifração autenticada em sessões. Cada instância pode ter múltiplas construções *duplex* internas, denominadas ***Pistons***, que são gerenciadas pela ***Engine***, como desmontrado na **Figura 2**.

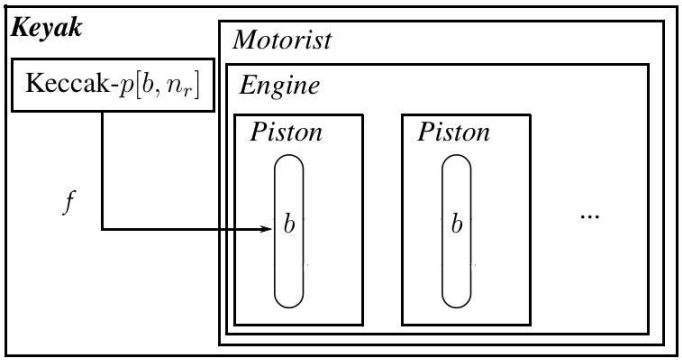

Figura 2. Diagrama esquemático do *Keyak*. A função de permutação f das esponjas é a *Keccak-$p[b, n_r]$*, de mesma largura b que as esponjas internas do *Motorist*.

4.1 Motorist

O papel do *Motorist* é cifrar ou decifrar mensagens, que são compostas de texto claro e metadados, e um possível rótulo (*tag*) associado para ser verificado. Note que é possível enviar somente texto claro, somente metadados ou ainda uma mensagem nula, apenas para verificar um rótulo fornecido.

A mensagem é distribuída entre os vários *Pistons*, cada qual recebendo uma parte diferente da mensagem. Para garantir que as esponjas dependam da mensagem inteira, o *Motorist* realiza o chamado ***knot***, em que concatena parte do estado de todos os *Pistons* em uma mensagem resultante que é inserida por inteira em cada *Piston*.

Uma sessão pode ser iniciada com uma *Session Unique Value*(SUV) composta de uma **chave secreta** e de um ***nonce***. A chave possui um tamanho máximo, porém não há limite para o tamanho do *nonce*. Uma interface de referência [J. Wetzels et al 2015] para o *Motorist* é:

$$\text{Motorist}[f, \Pi, W, c, \tau]$$

Sendo f a função de permutação a ser aplicada nos estados internos dos *Pistons*, Π o número de *Pistons*, W a unidade de alinhamento para o estado, capacidade c e tamanho τ de rótulo. *Motorist* possui quatro rotinas, sendo elas:

- *Motorist.StartEngine*: Inicializa a sessão com um *SUV* fornecido.
- *Motorist.Wrap*: Cifra ou decifra uma mensagen fornecida.
- *Motorist.MakeKnot*: Realiza um *knot* com o estado atual de todos os *Pistons*. No caso de haver mais de um *Piston*, ou a propriedade de esquecimento da sessão for requisitada, este procedimento será executado pelas rotinas *Motorist.StartEngine* e *Motorist.Wrap*.
- *Motorist.HandleTag*: Produz um novo rótulo ou verifica um rótulo fornecido. No caso de um novo rótulo ter sido requisitado ou um ter sido requisitada a validação de um rótulo fornecido, este procedimento será executado pelas rotinas *Motorist.StartEngine* e *Motorist.Wrap*.

4.2 Engine

Engine é uma construção suporte para o *Motorist*, responsável por gerenciar todos os *Pistons*, separando de maneira correta os dados de entrada e concatenando as várias saídas de cada *Piston*. Uma interface de referência [J. Wetzels et al 2015] para a *Engine* é:

$$\text{Engine}[\Pi, \textit{Pistons}]$$

Sendo Π o número de instâncias no vetor *Pistons*. *Engine* possui cinco rotinas, sendo elas:

- *Engine.Crypt*: Induz todo *Piston* a cifrar ou decifrar dados da entrada e retornar a saída correspondente.
- *Engine.Inject*: Induz todo *Piston* a injetar metadados da entrada.
- *Engine.Spark*: Induz todo *Piston* a aplicar a função de permutação f.
- *Engine.GetTag*: Induz todo *Piston* a extrair parte de seu estado para ser usado como rótulo.
- *Engine.InjectCollective*: Induz todo *Piston* a injetar dados de entrada de forma integral, sem particionar entre os Π *Pistons*.

4.3 Piston

Piston é uma construção *duplex* do tipo *FKSD* com um estado de largura b *bits*, equipado com uma função de permutação f. O estado é inicialmente zerado, até ser inicializado com uma *SUV*. Uma interface de referência [J. Wetzels et al 2015] para o *Piston* é:

$$\text{Piston}[f, r_s, r_a]$$

Sendo f a função de permutação a ser aplicada no estado, r_s o índice da taxa de compressão e r_a o índice da taxa de absorção, com $r_s \leq r_a < b$.

O estado deve ser preenchido com dados do texto claro até o indice r_s, e depois preenchido com metadados até o indíce r_a. Os *bytes* restantes correspondem à capacidade, e alguns são utilizados para armazenar informações do estado como a *flag* de fim de mensagem (EOM), o índice do estado contendo o último *byte* inserido (*Crypt End*), e o início e fim da injeção (*Inject Start* e *Inject End*) que indicam o início e o fim do trecho de *bytes* inseridos de metadados no estado.

Sempre que o *Piston* tiver o estado inteiro preenchido por dados, ele deverá aplicar a função de permutação f e voltar a absorver mais dados. O *Piston* possui quatro rotinas, sendo elas:

- *Piston.Crypt*: Cifra texto claro inserindo o criptograma resultante no estado, ou decifra criptograma inserindo o texto claro resultante. Em ambos os casos o indíce *Crypt End* é atualizado informando onde foi o último *byte* inserido, que será sempre menor que r_s.
- *Piston.Inject*: Insere metadados no estado, atualizando os indíces *Inject Start* e *Inject End*. A quantidade inserida pode ser no máximo r_a.
- *Piston.Spark*: Aplica a função de permutação f no estado, e atualiza a *flag* de *EOM* indicando se a mensagem terminou ou não.
- *Piston.GetTag*: Extrai parte do estado para ser usado como rótulo de autenticação.

5. Implementação

As rotinas do *Motorist* e *Engine* são de controle e são executadas poucas vezes durante uma sessão, portanto otimizá-las resultaria em um pequeno ganho de desempenho. De fato, a própria interface de implementações no repositório oficial [G. Bertoni et al. 2015d] recomenda apenas que as **funções que interagem diretamente com o estado interno** e a ***Keccak-***p em si sejam implementadas de maneira otimizada. São elas:

- *Initialize*: Inicializa o estado interno com *bytes* zero.
- *AddByte* e *AddBytes*: Adiciona *bytes* no estado interno através da operação **XOR**.
- *OverwriteBytes* e *OverwriteWithZeroes*: Sobrescreve *bytes* no estado interno com *bytes* zero ou com *bytes* fornecidos.
- *ExtractBytes* e *ExtractAndAddBytes*: Extrai *bytes* do estado interno, podendo também adicionar *bytes* ao mesmo tempo.
- *Permute*: A função de permutação *Keccak-*p.

Tais funções foram então implementadas em ***ARM assembly*** e também em ***NEON intrinsics*** [ARM Holdings 2015]. Como as instâncias nomeadas fazem uso de ambas ***Keccak-***$p[800]$ e ***Keccak-***$p[1600]$, foi necessário implementá-las para cada especificação, totalizando quatro implementações de alto desempenho.

Keyak e *Motorist* foram implementadas de maneira integral como mostrado na teoria, com exceção de alguns paramêtros que foram fixados por serem comuns entre as instâncias nomeadas, como o número n_r de rodadas e tamanho τ do rótulo. A entidade *Engine* foi separada em duas entidades de acordo com o tamanho de *bits* com qual ela

opera, a fim de evitar executar operações de condição (*branches*). Já a entidade *Piston* foi implementada dentro da entidade *Engine* para diminuir o número de chamadas internas, e facilitar uma possível implementação paralela futura.

Para verificar a correção do código, foi usado uma mesma **bateria de testes** fornecida pelo código de referência dos autores do algoritmo [G. Bertoni et al. 2015c]. Da mesma forma, foi integrado um **código já existente** de *Assembly* de *ARMv7* no projeto [G. Bertoni et al. 2015d], a fim de verificação de desempenho.

O algoritmo foi implementado **a partir do zero** em ***C***, devido ao maior desempenho da linguagem e fácil integração com linguagem de montagem (*assembly*). Todas as implementações foram compiladas com *flag* de otimização ***O2*** em todos os arquivos fonte, tanto *C* quanto *assembly*.

O código produzido em linguagem de montagem *ARMv8* utilizou dos novos recursos da arquitetura ***AArch64***, como o maior número de registradores vetoriais e as novas operações do pacote de instruções ***SIMD***, como transposição de vetores para otimizar movimentação de dados entre registradores. Cada registrador vetorial possui tamanho de 128 *bits* o que permite guardar quatro e duas palavras quando a largura b for 800 e 1600 respectivamente. Os registradores gerais possuem 64 *bits*, podendo suportar uma palavra por registrador no caso de $b = 1600$, que não era possível em registradores de 32 *bits* em *ARMv7*.

6. Resultados

Os resultados mostrados na **Tabela 1** foram obtidos por média simples de cem execuções para cada especificação. Os números de ciclos foram obtidos utilizando operações em linguagem de montagem específicas para cada arquitetura. As implementações para *ARMv7* foram executadas em um *Cortex-A8* e as implementações para *ARMv8* foram executadas em um *Cortex-A53*. Ambos dispositivos executavam sistema operacional ***ArchLinux***, o que permitiu compilar nativamente as implementações.

Os resultados da instância nomeada *Lunar Keyak* foram omitidas por serem de ordem muito maior que as outras e por apresentarem diferenças de desempenho entre implementações redundantes com as outras instâncias. Observa-se pela **Tabela 1** que o algoritmo fica naturalmente mais rápido sendo implementado em uma arquitetura *64 bits* como *ARMv8* do que em uma arquitetura *32 bits* como o *ARMv7*, como é possível observar na redução de número de ciclos na implementação em *C* do *ARMv7* para o *ARMv8*.

As implementações em *C* e *NEON intrinsincs* para ambas arquiteturas foram desenvolvidas neste trabalho, junto com a implementação *assembly* para *ARMv8*, aparecendo em itálico na **Tabela 1**. **A implementação *assembly* para *ARMv7* foi retirada do código de referência dos autores [G. Bertoni et al. 2015d] para nível de comparação**, e aparece em negrito na **Tabela 1**.

Tabela 1. Desempenho entre implementações - *Keyak Wrap*

River Keyak			
Processador	Implementação	Ciclos/*byte*	Ganho
ARMv7	*C*	249.88	-
	NEON Intrinsics	237.42	5,25%
	Assembly	26.87	829,95%
ARMv8	*C*	232.05	-
	NEON Intrinsics	63.45	256,72%
	Assembly	36.87	529,37%

Lake Keyak			
Processador	Implementação	Ciclos/*byte*	Ganho
ARMv7	*C*	165.35	-
	NEON Intrinsics	84.53	95,61%
	Assembly	15.10	995,03%
ARMv8	*C*	106.12	-
	NEON Intrinsics	19.19	453,00%
	Assembly	15.52	583.76%

Sea Keyak			
Processador	Implementação	Ciclos/*byte*	Ganho
ARMv7	*C*	178.20	-
	NEON Intrinsics	93.89	89.79%
	Assembly	15.97	1015.84%
ARMv8	*C*	116.03	-
	NEON Intrinsics	20.27	472.42%
	Assembly	16.37	608.79%

Ocean Keyak			
Processador	Implementação	Ciclos/*byte*	Ganho
ARMv7	*C*	202.29	-
	NEON Intrinsics	106.68	89.62%
	Assembly	17.79	1037.09%
ARMv8	*C*	132.45	-
	NEON Intrinsics	22.78	481.43%
	Assembly	18.45	617.89%

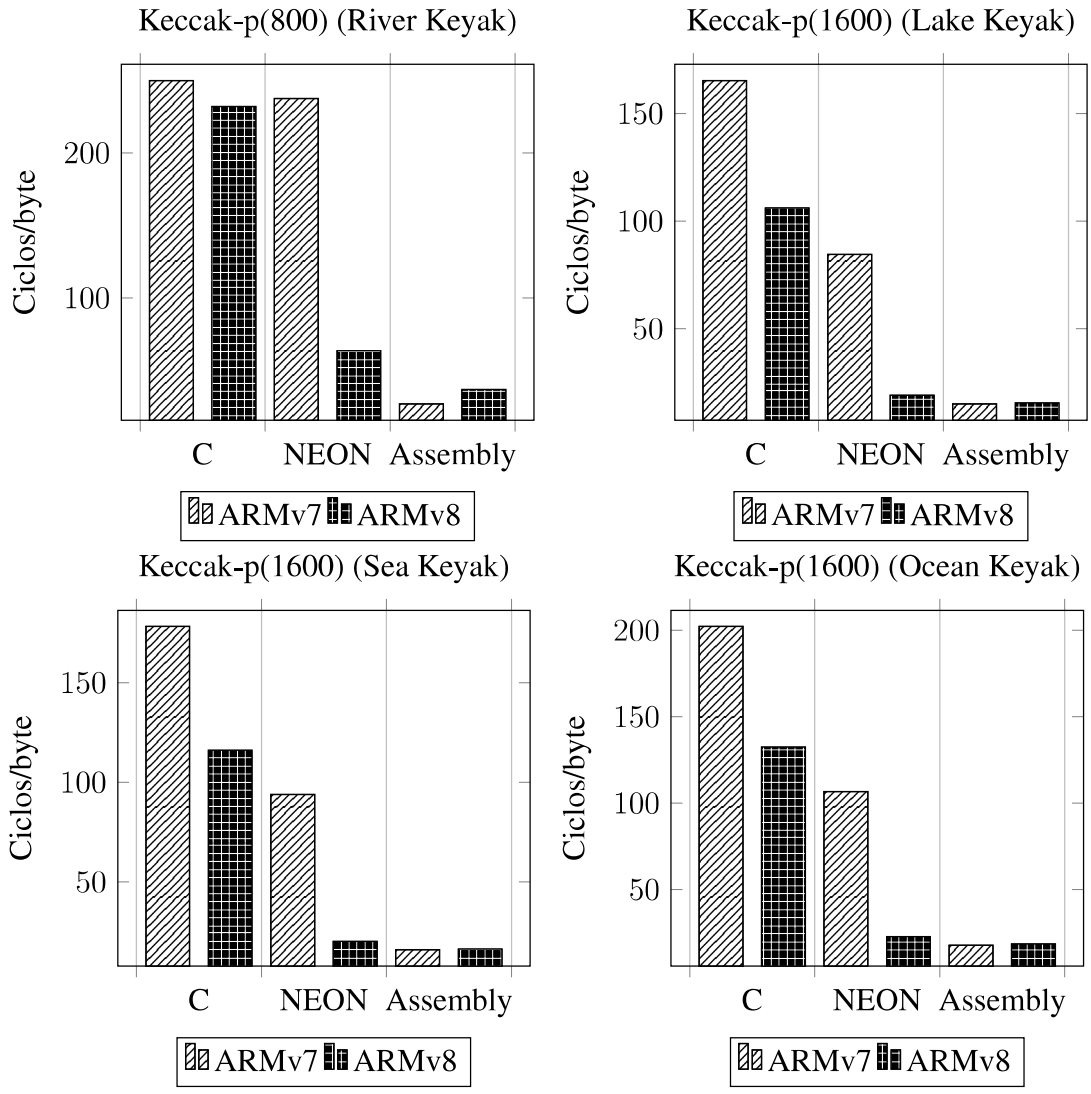

Figura 3. Gráficos dos resultados da Tabela 1.

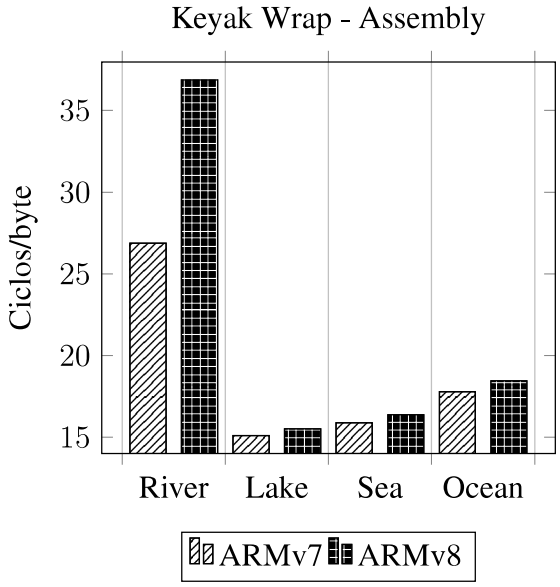

Figura 4. Gráfico comparando implementações em linguagem de montagem.

7. Conclusão

Os resultados da **Figura 3** mostram que a implementação em *NEON Intrinsics*, comparado à implementação em *C*, mostrou um alto desempenho em *ARMv8* nas instâncias que utilizam da *Keccak-*$p[1600]$, porém o desempenho em *ARMv7* foi relativamente inferior. Os resultados comprovam a melhor eficiência do código escrito em *NEON Intrinsics* para *ARMv8*. Já a instância que utiliza da *Keccak-*$p[800]$, *River Keyak*, teve uma melhora insignificativa em *ARMv7* e regular em *ARMv8*.

Para a implementação em código *assembly*, os resultados da **Figura 4** mostram que o desempenho obtido em *ARMv8* foi sempre marginalmente pior do que em *ARMv7* para as instâncias que utilizam da Keccak-$p[1600]$. Novamente na *River Keyak*, é possível perceber que ambas implementações possuem desempenho ineficiente, mas com a implementação em *ARMv8* gastando quase 40% mais ciclos que a de referência em *ARMv7*, apesar de ainda ser melhor do que as implementações em *C* e a *NEON Intrinsics* para *ARMv8*.

Como dito anteriormente na **Seção 2**, a implementação de referência [G. Bertoni et al. 2015d] utiliza do conjunto de instruções vetoriais apenas para a Keccak-$p[1600]$, sendo a Keccak-$p[800]$ implementada apenas com instruções regulares de montagem da arquitetura *ARMv7*, uma vez que é projetada para dispositivos com recursos limitados que não possuem suporte a *NEON*. As operações vetoriais acabam não sendo eficientes na Keccak-$p[800]$ devido à constante manipulação de palavras para montar os registradores vetoriais.

Assim, é possível que uma implementação que **não** use de operações vetoriais para Keccak-$p[800]$ tenha desempenho superior comparada à desenvolvida neste trabalho. Resta também implementar rotinas específicas para as instâncias com múltiplas esponjas, utilizando **paralelização** para otimizar seu desempenho. Tais tarefas podem vir a ser realizadas em trabalhos futuros.

Referências

G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche (2011). *Cryptographic sponge functions*. http://sponge.noekeon.org/

G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche (2013). *The Keccak sponge function family*. http://keccak.noekeon.org/

G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche (2013). *The Keccak implementation overview*. http://keccak.noekeon.org/Keccak-reference-3.0.pdf

G. Bertoni, J. Daemen, M. Peeters, G. Van Assche, and R. Van Keer (2015). *Keyak Specification*. http://keyak.noekeon.org/

G. Bertoni, J. Daemen, M. Peeters, G. Van Assche, and R. Van Keer (2015). *Keyakv2 CAESAR Submission*. http://keyak.noekeon.org/Keyak-2.0.pdf

G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche (2011). *Duplexing the sponge: single-pass authenticated encryption and other applications*. Selected Areas in Cryptography (SAC).

G. Bertoni, J. Daemen, M. Peeters, G. Van Assche, and R. Van Keer (2015). *Keccak Tools software*. https://github.com/gvanas/KeccakTools

G. Bertoni, J. Daemen, M. Peeters, G. Van Assche, and R. Van Keer (2015). *Keccak Code Package*. https://github.com/gvanas/KeccakCodePackage

National Institute of Standards and Technology (2014). *SHA-3 Standard: Permutation-Based Hash And Extendable-Output Functions*. http://www.nist.gov/manuscript-publication-search.cfm?pub_id=919061

J.P. Aumasson and D. Khovratovich (2009) *First analysis of Keccak*. http://131002.net/data/papers/AK09.pdf

K. Graves (2010) *Certified Ethical Hacker Study Guide, 1st Edition*.

F. Callegati, W. Cerroni and M. Ramilli (2009). *IEEE Xplore - Man-in-the-Middle Attack to the HTTPS Protocol*

J. Wetzels and W. Bokslag (2015) *Sponges and Engines: An introduction to Keccak and Keyak*.

K. Paterson (2013) *Authenticated Encryption in TLS*.

H. Krawczyk (2001) *The Order of Encryption and Authentication for Protecting Communications (Or: How Secure is SSL?)*.

B. Mennink, R. Reyhanitabar, and D. Vizar (2015) *Security of Full-State Keyed and Duplex Sponge: Applications to Authenticated Encryption*.

ARM Holdings (2015) *A64 Advanced SIMD Vector Instructions*. http://infocenter.arm.com

GitHUB (2016) *KeccakP-800 and KeccakP-1600 implementation for the ARM AArch64*. https://github.com/gvanas/KeccakCodePackage/pull/23

GitHUB (2016) *KeccakP1600 ARMv7A possible bug*. https://github.com/gvanas/KeccakCodePackage/issues/22

